Best Known Method:
Dealing With Long Diagnostic Lists in Intel® Thread Checker

By Rich Gerber and Clay Breshears

▪ What is the problem and what is a long Diagnostic List anyway?

The screenshot shown in Figure 1 is the Intel® Thread Checker Diagnostic List from a fairly short application – only 2500 lines of code – but more than 100 issues appear in the list.

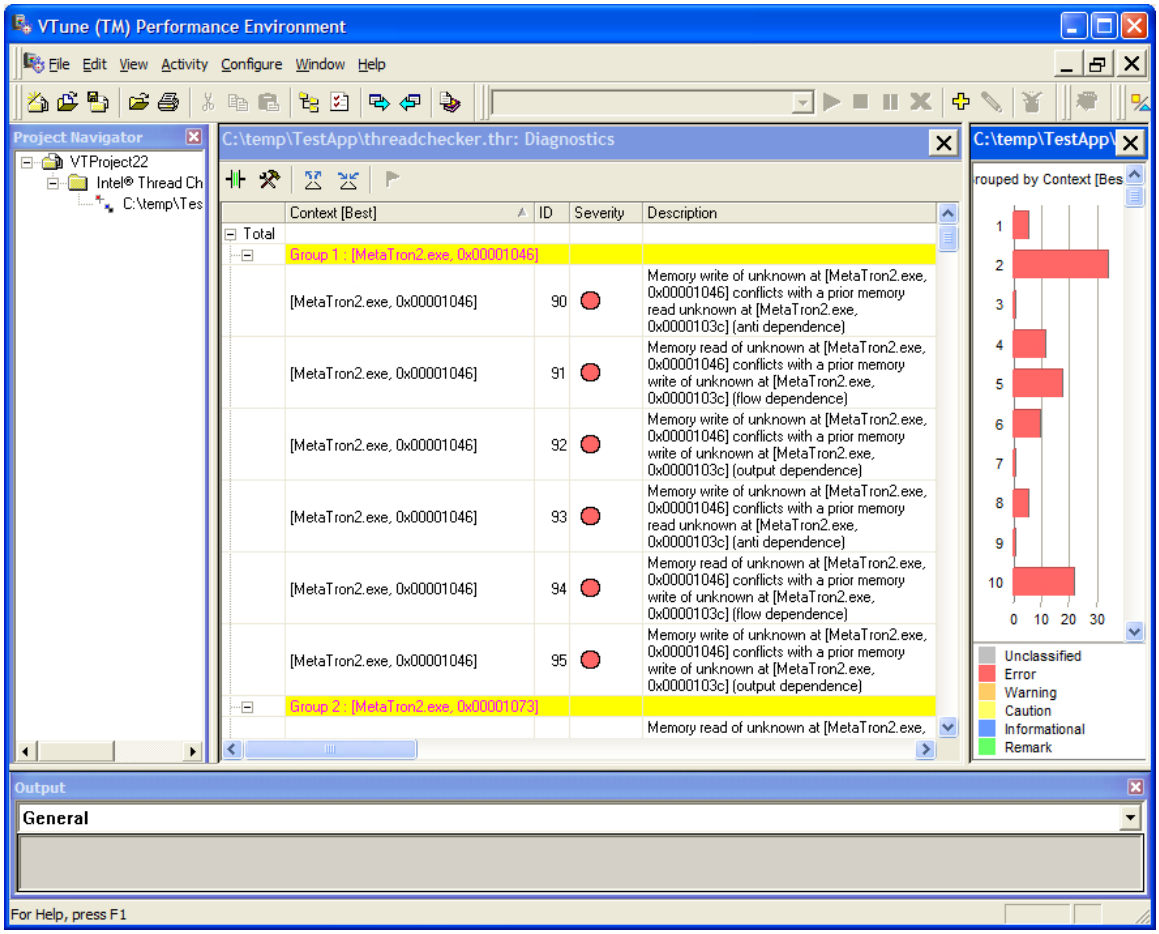

Figure 1 Long Diagnostic List

The problem is: Where to begin the debugging? Are all the diagnostic messages equally serious? Are any caused by other problems or the same problem?

For the engineer too short on time to read a five page document, the recommended solution is to group by terse description, sort by 1st Access [Best], treat diagnostic messages that occur on the same line of code as the same issue, and work on the Write->Write diagnostic messages first.

▪ What the Thread Checker shows you by default.

The initial Thread Checker view shows the Project Navigator on the left, the Diagnostic list in the middle, a bar graph summary of the diagnostics on the right, and the output window on the bottom. In Figure 1, the diagnostics are ‘grouped by context’, which is mostly equivalent to saying ‘grouped by function’. You can see in the bar graph that some functions contain many more diagnostics than others. In Figure 1, the function corresponding to the second group leads the pack with 34 diagnostics while the third, seventh, and ninth groups contain only a single diagnostic each. The functions appearing in the Diagnostic list are not the only functions in the application, just those for which Thread Checker identified an error or point of concern while running the present dataset. Functions that did not generate diagnostic messages are not listed.

The description column contains details about a diagnostic found on a specific line of source code. Most often these will be data races between threads. Double-clicking on a diagnostic will open a pane that allows you to view the lines of source code corresponding to that diagnostic. The same line of code may be listed multiple times when the data referenced on that source line is involved in storage conflicts among multiple threads executing at other points within the program.

Finally, when a thread exits, the Thread Checker captures and displays the stack usage information. This information is useful when debugging thread stack overflows and when analyzing and optimizing memory usage.

▪ Massaging the display to determine what to analyze first.

Remember, all diagnostics should be examined because every one may be serious. You could simply start at the beginning and examine each one in order. However, that approach is likely to include some redundant work as multiple issues involving the same memory locations may be listed multiple times yet require only a single change to correct.

A better approach is to sort the display so that the most serious diagnostics, and multiple diagnostics involving the same variable, will appear together. Start by opening the Column Configuration Dialog Box and selecting Terse mode as shown in Figure 2.

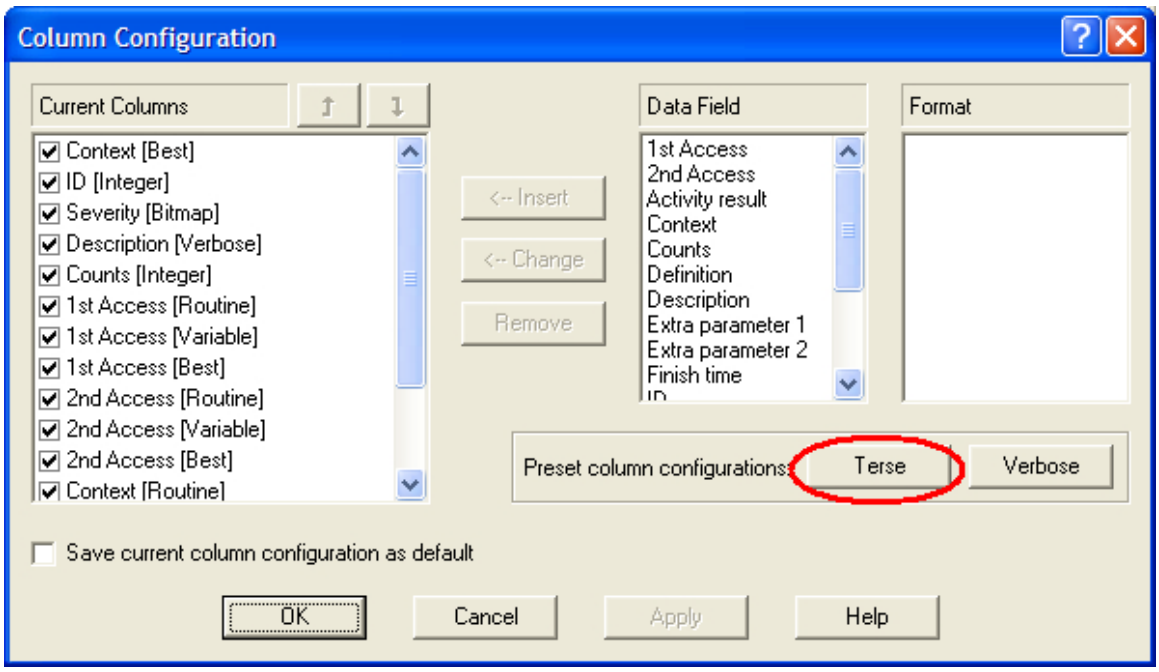

Figure 2 Configure Columns dialog box

Now with Terse mode selected, group the Diagnostics list by Description using the Group By dialog box shown in Figure 3

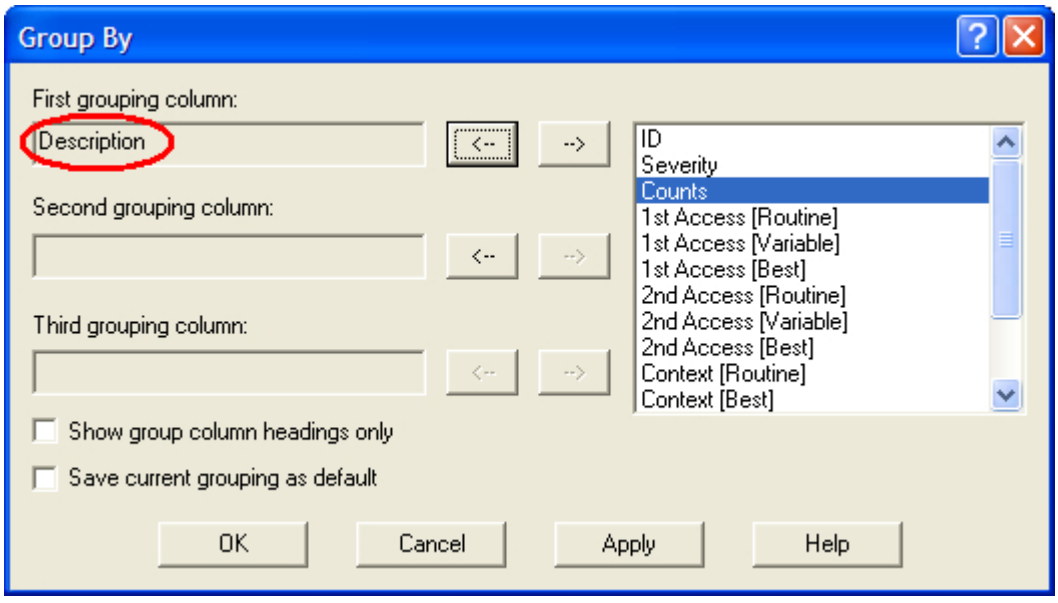

Figure 3 Group first column by Description

Now back at the Diagnostic list, click on the 1st Access [Best] column to sort by variable name. The diagnostics will now be grouped by type and then, in each subgroup, multiple entries for the same variable will appear together as shown in Figure 4. At this point, the bar graph on the right side shows a histogram of the number of diagnostics per type.

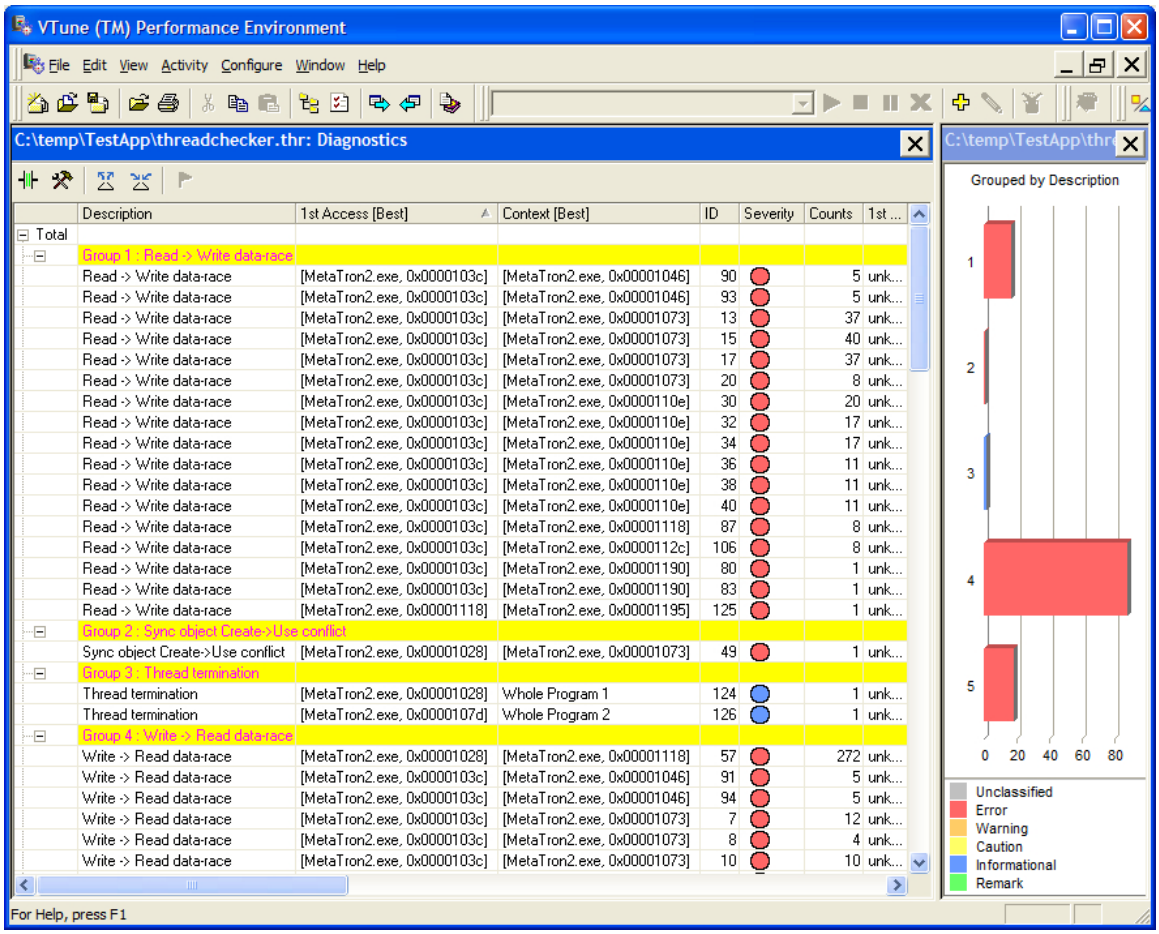

Figure 4 Diagnostics grouped by terse description and sorted by 1st Access [Best] with Navigator and Output window closed

▪ Analyzing the Diagnostics

Now that the diagnostics are grouped by Description and sorted by 1st Access [Best], it is time to begin your analysis. Start by looking at the Write -> Write date-race group of diagnostics first. These diagnostics are usually easiest to understand and fix and have the highest likelihood of causing additional diagnostics found in the other error groupings. Since diagnostics that occur on the same line of source or nearby lines are commonly fixed with a single modification, be sure to analyze the Diagnostic list in groups according to the 1st Access [Best] column. Usually declaring an operation to be atomic, making a variable private to a thread, updating the variable within a critical section, or using synchronization to control thread access will solve the error. But before jumping right in and attempting to fix an error, think about the root cause of the problem and solve it in the most efficient manner possible. Occasionally, a slightly 'larger' change to an algorithm will produce faster, more readable code.

Once you have worked through some of the Write -> Write date-race diagnostics, rerun the software through Thread Checker to see if your solutions were correct and complete and you have not introduced more errors. Also, you may find that other diagnostics have disappeared due to the changes you made when correcting the Write -> Write date-race

diagnostics. Continue fixing the Write -> Write date-race diagnostics then move on to the Write -> Read date-race diagnostics, then any remaining diagnostic groups. Be sure to rerun Thread Checker on the software after you have fixed a number of diagnostics. This will ensure you are making progress and save time by not worrying about diagnostics that are corrected by some prior fix.

- **Keep-in-mind…**

This document describes only *one* method for working with the diagnostic list, not the *only* method. Don't be shy; feel free to experiment with Thread Checker to find a method that works best for you!